EXPRESS rule writing

Lillehammer

Sunday 6th June 1999

Agenda

- Rule selection
- Rule writing
- Common problems
- Questions and answers

NOTICE

- Usage of examples from current STEP parts is for instructional purposes only, there are no value judgements associated with these selections. However, where parts have used invalid EXPRESS these errors have been reported.
- This course uses the concepts from the proposed ISO 10303-11TC2, which was produced to resolve ambiguities in the EXPRESS language. In many cases the common problems we discus are due to ambiguities in the EXPRESS language, not poor modelling.

Acknowledgements

- Thanks to the following for providing example problems:
  - Markus Maier
  - Jorulv Rangnes

Requirements

- Idea is to have rules which are:
  - Parsable by computers
  - Understandable by humans
- To support both requires a formal language and consistent style of use.

Requirements

- All constraints in an information model should be both parsable by computers and understandable by humans. The algorithmic nature of rules is easily parsable by computers but without a consistent style is difficult for human understanding. This session therefore will attempt to explain the style of rule writing used within SC4.

Rule selection

- Which rule do we choose?
  - where rule
  - unique rule
  - global rule

Where rules

- Used for constraints to be tested for each and every instance of an entity or defined type.

Unique rules

- Used for constraints on the whole population of an entity when constraint is based upon unique values for attributes of that entity or it’s supertypes.

Global rules

- Used when constraint cannot be specified by the previous two methods.

Example 1

- Constrain the following such that there can only be one instance of employee with position of general_manager.

```
TYPE job_title = ENUMERATION OF (general_manager, manager, worker);
END_TYPE;

ENTITY employee;
  name     : person_name;
  position : job_title;
END_ENTITY;
```

eurostep

Example 1: Answer

- We cannot use a local rule since we need access to all instances (to check there is only one).
- We cannot use a unique rule on position since this would also constrain there to be only one manager and one worker.
- Therefore we must write a global rule to check this constraint.

Example 2

- Constrain the following such that no two cars can have the same serial number if they are manufactured by the same manufacturer.

```
ENTITY car;
  manufactured_by : manufacturer;
  serial_number   : INTEGER;
END_ENTITY;
```

Example 2: Answer

- We cannot use a local rule since we need access to all instances.
- We cannot use a unique on serial number since this would constrain all manufacturers to use unique serial numbers.
- We can use unique on the combination of serial number and manufacturer.
- We could write a global rule, but this constraint is easier stated in a unique clause.

Example 3

- Constrain the following such that only valid dates are allowed (ignoring leap years).

```
ENTITY date;
  day : INTEGER;
  month : INTEGER;
END_ENTITY;
```

Example 3: Answer

- Since all the attributes required are local, and we don’t need access to each instance individually this can be written using a WHERE rule.

Formulating where rules

- simple arithmetic expression
- function call
- structured aggregate test
- Dealing with indeterminate (?) values

Formulating where rules

- Uses of the where rule:
- Domain constraint, ensuring valid values for attributes or combinations of attributes.
- Aggregate element constraints, ensuring all elements of an aggregate obey a constraint.
- Context constraint, ensuring that the type of an attribute is constrained within a AP.

Simple arithmetic expression

- Example 3 could be written as a domain constraint as follows:

```
ENTITY date;
  day   : INTEGER;
  month : INTEGER;
WHERE
  valid_month : { 1 <= month <= 12 } ;
  valid_day   : valid_date(day, month);
END_ENTITY;
```

- This relies on a function call to calculate if a date is valid or not with respect to the month.

Function call

```
FUNCTION valid_date (day, month : INTEGER) : BOOLEAN;
IF NOT {1 <= month <= 12}
THEN
  RETURN FALSE;
ELSE
  CASE month OF
      4, 6, 9, 11                  : RETURN ( {1 <= day <= 30} );
      2                            : RETURN ( {1 <= day <= 28} );
    OTHERWISE                      : RETURN ( {1 <= day <= 31} );
  END_CASE;
END_IF;
END_FUNCTION;
```

eurostep

Structured aggregate test

- Based upon the following:
- SIZEOF function;
  - Returns the size of any aggregate.
- QUERY expression.
  - Returns an aggregate which contains those elements from the input aggregate which pass a specified test.

Structured aggregate tests

- SIZEOF(QUERY (...)) = 0
  - There shall be none of the elements for which this test is true.
- SIZEOF(QUERY (...)) > 0
  - There shall be at least one element for which the test is true.
- SIZEOF(QUERY (...)) = x
  - There shall be x elements for which the test is true.

Structured aggregate tests: example

- The EXPRESS answer for example 1 earlier is:

```
RULE one_general_manager FOR (employee);
WHERE
  SIZEOF( QUERY( emp <* employee | emp.position = general_manager ) ) = 1;
END_RULE;
```

- “employee” is the set of all employee instances within the system.
- The QUERY results in a set of general_managers.
- SIZEOF() = 1, means there is only one.

Structured aggregate: Exercise

- Write a rule to ensure that there are containers that are loaded with sugar in the population of ships.

```
ENTITY ship;                             ENTITY container;
  load : SET OF container;                 content: cargo;
END_ENTITY;                              END_ENTITY;
TYPE cargo = ENUMERATION OF ( sugar, flour, empty );
END_TYPE;
RULE sugar_carriers FOR (ship);
 WHERE ????
```

eurostep

Structured aggregate: Exercise answer

```
RULE sugar_carriers FOR(ship);
  WHERE
  WR1: SIZEOF(QUERY(a_ship <* ship |
                                        SIZEOF(QUERY(a_container <* a_ship.load |
                                                             a_container.content = sugar
                                        ))>0
             ))>0;
END_RULE;
```

Dealing with indeterminate (?) 1

- May have optional attributes within an entity or optional elements in an array.
- May be due to attributes not being present as expected in instances.
- Need to be able to test if these exist (have a value) before using them or provide a default value.

Dealing with indeterminate (?) 2

- EXPRESS provides the EXISTS function for this purpose.
- EXPRESS also provides NVL to allow usage of default values if source is indeterminate.
- Indeterminate in a logical expression is handled as UNKNOWN
- This leads to the following rule format:

```
( EXISTS(A) AND ( (* tests using A*) ) OR NOT EXISTS(A)
```

Reading complex rules

- Use of a logical approach
- Some examples

Use of a logical approach

- Most rules can be treated as Logic predicates

```
wr1: SIZEOF(QUERY ( style1 <* SELF.styles | (NOT (SIZEOF( QUERY (
     style2 <* (SELF.styles - style1) | (NOT ((TYPEOF( style1) <>
     TYPEOF(style2)) OR (SIZEOF(['EXPLICIT_DRAUGHTING.' +
     'SURFACE_STYLE_USAGE', 'EXPLICIT_DRAUGHTING.' +
     'EXTERNALLY_DEFINED_STYLE'] * TYPEOF(style1)) = 1))) )) = 0)) )) =
     0;
```

Use of a logical approach

```
wr1: SIZEOF(QUERY ( style1 <* SELF.styles | (NOT (SIZEOF( QUERY (
      style2 <* (SELF.styles - style1) | (NOT ((TYPEOF( style1) <>
      TYPEOF(style2)) OR (SIZEOF(['EXPLICIT_DRAUGHTING.' +
      'SURFACE_STYLE_USAGE', 'EXPLICIT_DRAUGHTING.' +
      'EXTERNALLY_DEFINED_STYLE'] * TYPEOF(style1)) = 1))) )) = 0)) )) =
      0;
```

- intersect the type of style1 with a known set of type names

Use of a logical approach

```
wr1: SIZEOF(QUERY ( style1 <* SELF.styles | (NOT (SIZEOF( QUERY (
    style2 <* (SELF.styles - style1) | (NOT ((TYPEOF( style1) <>
    TYPEOF(style2)) OR (SIZEOF(['EXPLICIT_DRAUGHTING.' +
    'SURFACE_STYLE_USAGE', 'EXPLICIT_DRAUGHTING.' +
    'EXTERNALLY_DEFINED_STYLE'] * TYPEOF(style1)) = 1))) )) = 0)) )) =
    0;
```

- style1 is either a surface_style_usage or an externally_defined_style (but not both!)

Use of a logical approach

```
wr1: SIZEOF(QUERY ( style1 <* SELF.styles | (NOT (SIZEOF( QUERY (
    style2 <* (SELF.styles - style1) | (NOT ((TYPEOF( style1) <>
    TYPEOF(style2)) OR (SIZEOF(['EXPLICIT_DRAUGHTING.' +
    'SURFACE_STYLE_USAGE', 'EXPLICIT_DRAUGHTING.' +
    'EXTERNALLY_DEFINED_STYLE'] * TYPEOF(style1)) = 1))) )) = 0)) )) =
    0;
```

- style1 not the same as style2

Use of a logical approach

```
wr1: SIZEOF(QUERY ( style1 <* SELF.styles | (NOT (SIZEOF( QUERY (
      style2 <* (SELF.styles - style1) | (NOT ((TYPEOF( style1) <>
      TYPEOF(style2)) OR (SIZEOF(['EXPLICIT_DRAUGHTING.' +
      'SURFACE_STYLE_USAGE', 'EXPLICIT_DRAUGHTING.' +
      'EXTERNALLY_DEFINED_STYLE'] * TYPEOF(style1)) = 1))) )) = 0)) )) =
      0;
```

- style1 the same as style2 and type of style1 neither surface_style_usage nor externally_defined_style

Use of a logical approach

```
wr1: SIZEOF(QUERY ( style1 <* SELF.styles | (NOT (SIZEOF( QUERY (
    style2 <* (SELF.styles - style1) | (NOT ((TYPEOF( style1) <>
    TYPEOF(style2)) OR (SIZEOF(['EXPLICIT_DRAUGHTING.' +
    'SURFACE_STYLE_USAGE', 'EXPLICIT_DRAUGHTING.' +
    'EXTERNALLY_DEFINED_STYLE'] * TYPEOF(style1)) = 1))) )) = 0)) )) =
    0;
```

- There must not exist a style1 the same as style2 where the type of style1 is neither surface_style_usage nor externally_defined_style

Use of a logical approach

```
wr1: SIZEOF(QUERY ( style1 <* SELF.styles | (NOT (SIZEOF( QUERY (
    style2 <* (SELF.styles - style1) | (NOT ((TYPEOF( style1) <>
    TYPEOF(style2)) OR (SIZEOF(['EXPLICIT_DRAUGHTING.' +
    'SURFACE_STYLE_USAGE', 'EXPLICIT_DRAUGHTING.' +
    'EXTERNALLY_DEFINED_STYLE'] * TYPEOF(style1)) = 1))) )) = 0)) )) =
    0;
```

- There must exist a style1 the same as style2 where the type of style1 is neither surface_style_usage nor externally_defined_style

Use of a logical approach

```
wr1: SIZEOF(QUERY ( style1 <* SELF.styles | (NOT (SIZEOF( QUERY (
     style2 <* (SELF.styles - style1) | (NOT ((TYPEOF( style1) <>
     TYPEOF(style2)) OR (SIZEOF(['EXPLICIT_DRAUGHTING.' +
     'SURFACE_STYLE_USAGE', 'EXPLICIT_DRAUGHTING.' +
     'EXTERNALLY_DEFINED_STYLE'] * TYPEOF(style1)) = 1))) )) = 0)) )) =
     0;
```

- For each style in styles there shall not exist two styles the same when the type of the style is neither surface_style_usage nor externally_defined_style

Examples

```
wr1: SIZEOF(['EXPLICIT_DRAUGHTING.TEXT_LITERAL',
        'EXPLICIT_DRAUGHTING.ANNOTATION_TEXT',
        'EXPLICIT_DRAUGHTING.ANNOTATION_TEXT_CHARACTER',
        'EXPLICIT_DRAUGHTING.DEFINED_CHARACTER_GLYPH',
        'EXPLICIT_DRAUGHTING.COMPOSITE_TEXT']
       * TYPEOF(SELF\styled_item.item)) > 0;


wr1: ((NOT closed_curve) AND
(SIZEOF(QUERY ( temp <* segments | ( temp.transition = discontinuous) )) = 1))
OR (closed_curve  AND (SIZEOF(QUERY ( temp <* segments | (temp.transition
      =  discontinuous) )) = 0));
```

Writing functions

- Make sure all variables are initialised before use
- Make sure you deal with the unexpected, i.e. indeterminate values being passed as parameters or attributes of parameters.

Common problems

- The following are examples of problems which have been detected in SC4 parts.
- Corrections, if supplied, are in green

String manipulation1

- Often seen rules with typographical problems (line length) adding STRINGS:

```
wr1: SIZEOF(QUERY ( a <* approval | (NOT
        (SIZEOF(USEDIN(a, 'EXPLICIT_DRAUGHTING.' +
        'APPROVAL_ASSIGNMENT.' +
        'ASSIGNED_APPROVAL' )) >= 1)) )) = 0;
```

- What is being created here is the role name:

```
EXPLICIT_DRAUGHTING.APPROVAL_ASSIGNMENT.ASSIGNED_APPROVAL
```

eurostep

String manipulation2

- Here addition is incorrect, it should have been a ‘,’ as an element separator:

```
'request_date'      : IF SIZEOF (e.items) <>
                       SIZEOF (QUERY (x <* e.items |
                       SIZEOF (
                       ['CONFIG_CONTROL_DESIGN.CHANGE_REQUEST' +
                       'CONFIG_CONTROL_DESIGN.START_REQUEST'] *
                       TYPEOF (x)) = 1))
                       THEN RETURN(FALSE);
                     END_IF;
```

eurostep

String manipulation2

- Here addition is incorrect, it should have been a ‘,’ as an element separator:

```
'request_date'        : IF SIZEOF (e.items) <>
                         SIZEOF (QUERY (x <* e.items |
                         SIZEOF (
                         ['CONFIG_CONTROL_DESIGN.CHANGE_REQUEST',
                         'CONFIG_CONTROL_DESIGN.START_REQUEST'] *
                         TYPEOF (x)) = 1))
                         THEN RETURN(FALSE);
                       END_IF;
```

Group reference1

- Trying to access attributes not declared in the entity referred to by a group reference

```
 ENTITY composite_curve_on_surface
   SUPERTYPE OF(boundary_curve) SUBTYPE OF (composite_curve);
DERIVE
   basis_surface : SET[0:2] OF surface := get_basis_surface(SELF);
 WHERE
   …
END_ENTITY;
n := SIZEOF(c\composite_curve_on_surface.segments);
n := SIZEOF(c\composite_curve.segments); should have been used
```

Group reference2

- Returning a group reference from a function returns only a part of an instance!

```
RETURN (f\geometric_representation_item);
RETURN (f); should be used since f will be a geometric_representation_item
```

Bags and Sets

- Bags are not compatible with sets, you cannot assign a bag value to a set variable.

```
LOCAL
  x : SET OF symbol_representation_relationship;
END_LOCAL;
  x := USEDIN(relation\representation_relationship.rep_1,
               'REPRESENTATION_SCHEMA.'+
               'REPRESENTATION_RELATIONSHIP.'+ 'REP_2');
  x := bag_to_set(USEDIN(relation\representation_relationship.rep_1,
               'REPRESENTATION_SCHEMA.'+
               'REPRESENTATION_RELATIONSHIP.'+ 'REP_2'));
```

Role names

- Rolenames have a specific format as follows: `schema_name.entity_name.attribute_name`

```
ENTITY shape_representation_relationship
  SUBTYPE OF (representation_relationship);
WHERE
...
local_srr := local_srr +
bag_to_set(USEDIN(shape_representation_set[i],
                   'PRODUCT_PROPERTY_REPRESENTATION_SCHEMA.'+
                   'SHAPE_REPRESENTATION_RELATIONSHIP.REP_1'));
                   'PRODUCT_PROPERTY_REPRESENTATION_SCHEMA.'+
                   'REPRESENTATION_RELATIONSHIP.REP_1'));
```

Partial instantiation 1

- Entity assignment of requires valid complex instance

```
 ENTITY direction
   SUBTYPE OF (geometric_representation_item);
   direction_ratios : LIST [2:3] OF REAL;
 WHERE
...
v := direction([0.0,1.0,0.0]);
v := representation_item('') || geometric_representation_item() ||
       direction([0.0,1.0,0.0]);
```

The first part could be replaced by a constant as done now in 10303-42.

Partial instantiation 2

- Comparison of an instance with a partial complex entity value not defined

```
FUNCTION first_proj_axis
  (z_axis, arg: direction): direction;
...
IF z_axis <> direction([1,0,0]) THEN
IF z_axis\direction <> direction([1,0,0]) THEN
```

assuming we only wish to test the direction part.

Assignment to un-initialised variables 1

- The following is undefined

```
LOCAL
    result : direction ;
END_LOCAL;

   IF (vec.dim <> 2) OR NOT EXISTS (vec) THEN
     RETURN(?);
   ELSE
     result.direction_ratios[1] := -vec.direction_ratios[2];
```

Assignment to un-initialised variables 1

- It is defined only if we initialise the variable

```
LOCAL
  result : direction := representation_item('') ||
                        geometric_representation_item() ||
                        direction([0.0,0.0,0.0]);
END_LOCAL;

  IF (vec.dim <> 2) OR NOT EXISTS (vec) THEN
    RETURN(?);
  ELSE
    result.direction_ratios[1] := -vec.direction_ratios[2];
```

Assignment to un-initialised variables 2

```
LOCAL
  u : LIST [3:3] OF direction;
END_LOCAL;
u[3] := NVL(normalise(axis),direction([0,0,1]));
u[1] := first_proj_axis(u[3],ref_direction);
u[2] := normalise(cross_product(u[3],u[1])).orientation;
```

Assignment to un-initialised variables 2

```
LOCAL
  u1, u2, u3 : direction;
  u : LIST [3:3] OF direction;
END_LOCAL;
u3 := NVL(normalise(axis), representation_item('') ||
                         geometric_representation_item() || direction([0,0,1]));
u1 := first_proj_axis(u3,ref_direction);
u2 := normalise(cross_product(u3,u1)).orientation;
u := [u1, u2, u3];
```

Local variables not needed for repeat loops

- Increment control is implicitly declared

```
 LOCAL
  i : INTEGER;
END_LOCAL;

REPEAT i := 1 TO ndim;
   ...
END_REPEAT;
```

- Recommendation is to remove the local declaration of i.

Over to you!

- Formal part of this session is now finished
- For the remainder of the time we will work together to resolve problems you have in your schemas.